A Theoretical Standardized Assessment Framework for Single Variable Polynomial Root-Finding Algorithms

Luke Sellmayer | WRTG 3030 | November 16, 2023

Abstract

There is significant inconsistency in these papers when determining whether a new root-finding algorithm is superior to its predecessors. This paper develops a theoretical standardized testing framework measuring multiple metrics with standard calculation methods, which would enable a comprehensive evaluation of the performance of a given root-finding method.

Introduction

Numerous papers in the field of numerical analysis are dedicated to the development of novel iterative root-finding algorithms for polynomial functions, aiming to showcase the enhancements that these new methods offer compared to their predecessors. One such paper published in 1976 tested a family of related root-finding algorithms and identified one with a high accuracy of approximated roots with fewer iterations required compared to others in the family [3]. Another paper in 1994 introduced a method that exhibited lower CPU-time compared to previous methods [8]. More recently, Amir Naseem and colleagues demonstrated a method exhibiting fourth-order convergence, an improvement compared to similar methods in 2022 [9].

However, there is significant inconsistency in these papers when determining whether a new root-finding algorithm is superior to its predecessors. Some papers focus entirely on theoretical performance by analyzing the convergence rate. Others emphasize empirical performance by considering the CPU time or number of iterations required for the algorithm to converge to a root. The methods used to calculate convergence order vary, as do the tolerance levels used to determine when to terminate an algorithm. Moreover, many of these papers overlook another crucial aspect of their algorithm's performance: the robustness of the algorithm concerning its initial guess or initial interval input. This is of great practical significance, as it can be difficult to choose an initial guess that is close to the actual root or an initial interval containing the root; many of these algorithms only work when the above criteria are satisfied.

To facilitate a meaningful comparison of root-finding algorithms, the metrics used should be standardized and should encompass both theoretical and empirical aspects. This paper develops a theoretical standardized testing framework measuring multiple metrics with standard calculation methods, which would enable a comprehensive evaluation of the performance of a given root-finding method. Readers and researchers alike would then be able to use the framework to gain a more accurate picture of how a root-finding algorithm performs.

This paper will focus exclusively on iterative root-finding algorithms that deal with finding roots of single-variable polynomial functions; other types of single-variable functions, multi-variable functions, vector-valued functions, and systems of equations will not be considered.

Background

Prior to embarking on the development of this standardized assessment, it is crucial to conduct a comprehensive review of performance metrics and the methodologies employed in previous research. The objective of this assessment is to scrutinize the range of metrics used and subsequently select a subset that exerts the most significant influence on root-finding algorithm performance. Furthermore, examining the various methods used for calculating each metric will allow for the identification of a superior approach for the calculation of a given metric. Ultimately, this process will enable the establishment of standardized metrics and calculation methods that can be universally applied to any root-finding algorithm.

Convergence Order

Informally, convergence order describes how quickly a sequence approaches its limit; sequences with higher order convergence will take fewer iterations to reach their limit than sequences with lower order convergence. Formally, a sequence $\{p_n\}_{n=0}^{\infty}$ of root approximations that converges to the actual root r, with $p_n \neq r$ for all n, has convergence order α with asymptotic error constant λ, with $0 < \lambda < \infty$, if [1]

$$\lim_{n\to\infty} \frac{|p_{n+1} - r|}{(|p_n - r|)^{\alpha}} = \lambda.$$

Determining α can be done by defining the next iteration step as a function of the previous iteration, $p_{n+1} = g(p_n)$, and then performing a Taylor expansion of g centered at the root r to create the above expression; see [10] for more on how this is done. This method of determining convergence order is used in two papers [3][9].

Certain papers didn't explicitly employ the Taylor expansion method; instead, they assessed the convergence order by leveraging the fact that the newly proposed algorithm was a variation of another algorithm with a known convergence order [11][12]. Consequently, they could infer that the new algorithm would share the same convergence order as the original one. Another paper stated the convergence order of its algorithm but did not indicate how the order was calculated [8].

Methods for Convergence Order

Paper / **Method**	Hansen and Patrick	Lang and Frenzel	Yuksel	Naseem et al.	Thota
Taylor Series	X			X	
Variation of Another Algorithm			X		X
Unknown		X			

Number of Iterations and CPU Time

Many papers used either the number of iterations [3][9][11] or the CPU time [8][12] needed for an approximation to converge to a value within a specific tolerance to measure algorithm performance. However, there were differences in deciding when to terminate the algorithm. If the actual root of the polynomial was known, one could choose to terminate the algorithm once the error between an approximation at an iteration and the actual root was within a specific tolerance, used in one paper [11]. Another choice of termination is to do so when the error between successive iterations is within a specific tolerance, which tends to be used when the actual root of the polynomial is not known; this was used in two papers [3][9].

When CPU time was included as a metric, both papers using specified the hardware and software used to measure it [8][12].

Error Types

For both number of iterations and CPU time, the types of error used varied. Some used absolute error: between successive approximations p_n and p_{n-1}, absolute error is defined as

$$absolute\ error = |p_n - p_{n-1}|.$$

While for error between an approximation p_n and the actual root p^*, the absolute error is

$$absolute\ error = |p^* - p_n|.$$

Similarly, the relative error between successive approximations or between an approximation and the actual root is defined as

$$relative\ error = \frac{|p_n - p_{n-1}|}{p_n}$$

$$relative\ error = \frac{|p^* - p_n|}{p^*}$$

respectively.

Most papers did not specify which type of error was used.

Methods for Number of Iterations

Method | Error		Hansen and Patrick	Naseem et al.	Thota
Successive approximations	Absolute			
	Relative			
	Unknown	X	X	
Approximation and root	Absolute			X
	Relative			

	Unknown			

Methods for CPU Time			
Method | Error	Paper	Lang and Frenzel	Yuksel
Successive approximations	Absolute		
	Relative		
	Unknown		
Approximation and root	Absolute		
	Relative	X	
	Unknown		X

Tolerance Values

The value of the tolerance was also variable between papers; values ranged from $1 \cdot 10^{-1}$ all the way up to machine precision at $1 \cdot 10^{-16}$ [3][8][9][11][12]. Notably, some papers chose to include a range of tolerances while testing [11][12].

Tolerance Values Used					
Tol. Value \ Paper	Hansen and Patrick	Lang and Frenzel	Yuksel	Naseem et al.	Thota
$1 \cdot 10^{-1}$			X		
$1 \cdot 10^{-3}$			X		
$1 \cdot 10^{-4}$					X
$1 \cdot 10^{-5}$					X
$1 \cdot 10^{-7}$			X		
$1 \cdot 10^{-9}$					X
$* 1 \cdot 10^{-16}$		X	X		
Unknown	X			X	

Test Functions

Some papers used certain families of functions from specific engineering fields to test their algorithms. “High-Performance Polynomial Root Finding for Graphics” exclusively tested randomly generated Bernstein polynomials [12], which make up Bezier curves frequently used in typography and animation. Another paper from the IEEE Signal Processing Letters journal looked at polynomials containing “…a root distribution similar to the transfer function of an FIR low-pass filter…” [8]. The paper by Naseem et al. tested their algorithm using nonlinear equations from “real-world problems in engineering” such as blood rheology, fluid dynamics, and radiation [9].

Other papers did not seem to indicate how they decided which functions to test. While Naseem et al. also tested “five random problems” in addition to the equations from engineering contexts [9], Hansen and Patrick simply described their test functions as “arbitrarily chosen test polynomials” [3]; Thota only tested a single function with no explanation on why it specifically was used [11]. Only one paper included a non-engineering related test function while also justifying its inclusion, which was to “…check the performance of the programs for numerically well-conditioned but high-degree polynomials” [8].

Summary of Metrics Used

Metrics Used					
Paper / **Metric**	Hansen and Patrick	Lang and Frenzel	Yuksel	Naseem et al.	Thota
Convergence order	X	X	X	X	X
Number of iterations	X			X	X
CPU Time		X	X		
Approximation error		X			

The Theoretical Standard Testing Framework

With the review of metrics and methods done, we can now begin describing the theoretical standard testing framework. We’ll start by describing how test polynomials will be constructed, describe which metrics will be used and the methods used to calculate them, the actual process of the framework, and end with a proposal for how the framework could be implemented.

Test Polynomial Construction

As seen previously, the most glaring difference in how these algorithms are tested is the variety of polynomials used for testing. Multiple testing polynomials should be utilized in the standardized assessment framework to allow for a more comprehensive evaluation of a root-finding algorithm. Based on test polynomials used in previous research, the following properties should vary across testing polynomials used in the proposed testing framework:

- Form of polynomial

Polynomials can be written in standard form or factored form, so both should be utilized.

- Degree of polynomial

Some researchers may optimize their algorithm to work for polynomials within a certain range of degrees (CITE Yuksel). Polynomials of various degrees should be included.

- Entirely real or complex coefficient polynomials

Some researchers may only wish to run their algorithms on standard form polynomials with entirely real coefficients due to how such polynomials arise within certain contexts, such as computer graphics [12]. Both real and complex coefficient polynomials should be used.

- Polynomials with real or complex roots

Some researchers may only wish to find the real roots of polynomials. We need to include a few polynomials that have at least one real root.

- Multiplicities of polynomial roots

Some algorithms may be affected by the multiplicity of a root; famously, Newton's Method has quadratic convergence for simple roots, but only linear convergence for non-simple roots [2]. We should test multiple different multiplicity values for each factored form polynomial that we create.

- Randomization

Each specific polynomial should be somehow randomized, either by randomly generating the coefficients in standard form or the roots and multiplicities in factored form. This will ensure that researchers cannot optimize their algorithm for a specific polynomial ahead of time, as the exact characteristics of the polynomial will be different each time the standard assessment is run.

Standard Form Construction

Polynomials of degree n can be written in standard form,

$$P(x) = \sum_{j=0}^{n} a_j x^j = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

where the coefficients a_j may be either real or complex. By the fundamental theorem of algebra, every non-zero, single-variable polynomial of degree n with complex coefficients a_j has exactly n complex roots when counting multiplicities of non-simple roots. This means we are guaranteed that polynomials with degree $n \geq 1$ will have roots for a root-finding algorithm to find.

Creating a random polynomial with a given degree in standard form can be accomplished by randomly selected values for the real coefficients a_j from a uniform distribution on some interval $[\alpha, \beta]$. If the coefficients are complex, then $a_j = c + di$, and constants c and d can also be randomly selected from a uniform distribution on the same interval $[\alpha, \beta]$. One question that arises is deciding on the endpoints to use for the uniform distribution's interval; this paper suggests making the endpoints symmetrical and making sure to include 0 in the interval, letting $\alpha = -\beta$ and $\alpha < 0 < \beta$. The values chosen from $[\alpha, \beta]$ can be limited to integers, rationals, or reals for a given test polynomial.

We still need a method to generate entirely real coefficient polynomials that have at least one real root while still being randomized. One such method involves simply choosing the degree of the polynomial to be odd; this works because if the complex root $r = c + di$ with $d \neq 0$ is a root of the polynomial, then its complex conjugate $\bar{r} = c - di$ must also be a root. Therefore, there will always be an even number of complex roots with nonzero imaginary parts, meaning that for polynomials of odd degree, there must be at least one real root. While this method works regardless of the use of real or complex coefficients, this limits the degree of the polynomial to only be odd.

An alternative method that guarantees a polynomial has at least one real root works by ensuring that all roots of the polynomial are real. For a polynomial $P(x)$ in standard form of degree $n \geq 2$ with positive real coefficients a_j, if

$$a_j^2 - 4a_{j-1}a_{j+1} > 0 \quad \forall j \in \{1, 2, \dots, n-1\}$$

then all the roots of $P(x)$ will be real and distinct [5]. Using this method, we can generate randomized standard form polynomials with entirely real coefficients and entirely real roots by ensuring that the randomly generated coefficients satisfy the above conditions. While this method works for polynomials of any degree, we are limited to using only positive, real coefficients. Using both methods will ensure coverage of all possible polynomial degrees with both real and complex coefficients.

It is important to note that we cannot directly determine root values and their multiplicities when polynomials are in standard form, which motivates the need to also construct polynomials in factored form.

Factored Form Construction

In factored form, polynomials are written as

$$P(x) = \sum_{j=1}^{j_{max}} (x - r_j)^{m_j} = (x - r_1)^{m_1}(x - r_2)^{m_2} \dots \left(x - r_{j_{max}}\right)^{m_{j_{max}}}.$$

The roots of $P(x)$ can be directly read from this form, being $r_1, r_2, \dots, r_{j_{max}}$. We can randomly generate real roots by again randomly selecting values from a uniform distribution on some interval $[\alpha, \beta]$. For real roots, we can use the same uniform distribution that was used for determining the real coefficients a_j in standard form; for complex roots $r_j = c + di$ with $d \neq 0$, we can again select values for constants c and d from the same distribution. The same recommendations for the interval endpoints used in standard form apply here as well, and as before, the values chosen can be limited to integers, rationals, or reals.

The corresponding multiplicities of roots $r_1, r_2, \dots, r_{j_{max}}$ can also be read directly from this form, being $m_1, m_2, \dots, m_{j_{max}}$. These can also be randomly generated by selecting their values from a separate uniform distribution, this time on the interval $[1, n]$. For a given polynomial of degree n having n roots, we need to make sure that multiplicity values add up to n, i.e.

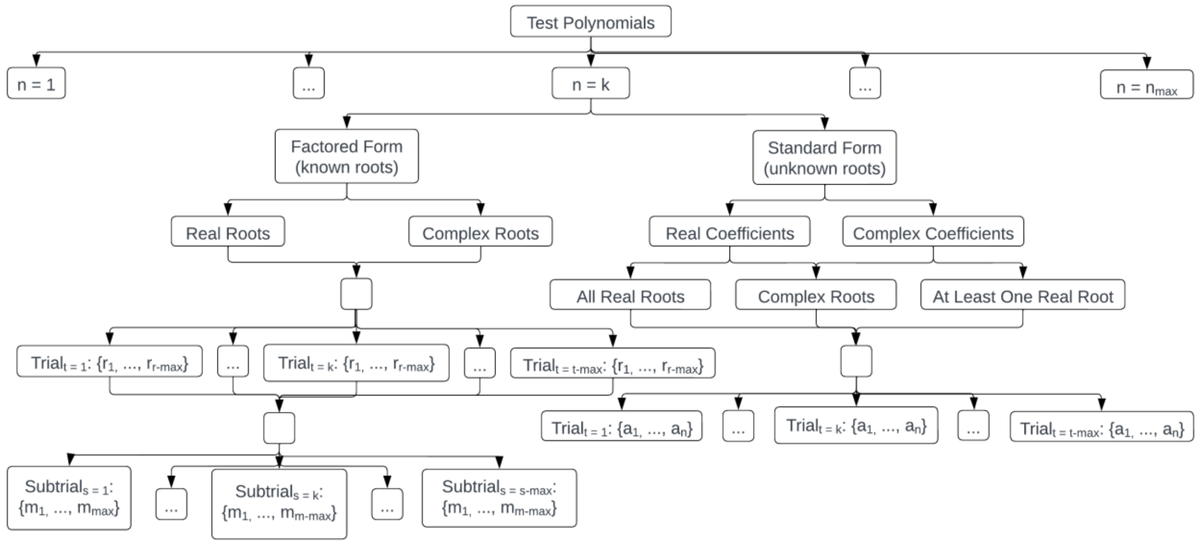
$$n = \sum_{j=1}^{j_{max}} m_j = m_1 + \cdots + m_{j_{max}}.$$

Summary

To summarize the polynomial construction method, polynomials will be constructed for all possible degrees up to some maximum degree, n_{max}. For each degree, polynomials in both standard form and factored form will be constructed.

In factored form, polynomials will be constructed to have either real roots or complex roots; for each factored form polynomial created, multiple trials, up to some max number, t_{max}, will be run, with each trial having randomly generated root values. Furthermore, each trial run will have sub-trials, with each sub-trial having randomly generated multiplicity values for the roots of the original trial.

In standard form, polynomials will be constructed to have either real or complex coefficients. Both real and complex coefficient polynomials will have constructions with complex roots and ones with at least one real root when the degree is odd; real coefficient polynomials will also have constructions that create entirely real roots, using the methods mentioned previously. For each standard form polynomial constructed, again, multiple trials, up to some max number, t_{max}, will be run, this time with each trial having randomly generated coefficient values.



Test Polynomials
n = 1
...
n = k
...
n = n_max
Factored Form (known roots)
Standard Form (unknown roots)
Real Roots
Complex Roots
Real Coefficients
Complex Coefficients
All Real Roots
Complex Roots
At Least One Real Root
Trial_t = 1: {r_1, ..., r_r-max}
Trial_t = k: {r_1, ..., r_r-max}
Trial_t = t-max: {r_1, ..., r_r-max}
Trial_t = 1: {a_1, ..., a_n}
Trial_t = k: {a_1, ..., a_n}
Trial_t = t-max: {a_1, ..., a_n}
Subtrial_s = 1: {m_1, ..., m_max}
Subtrial_s = k: {m_1, ..., m_m-max}
Subtrial_s = s-max: {m_1, ..., m_m-max}

Metrics to be Used

Condition Number

The condition number of a function measures how sensitive a function is to small changes in its input. For any single-variable function $f(x)$, such as a polynomial, the condition number $k(x)$ is given by

$$k(x) = \left|\frac{xf'(x)}{f(x)}\right|,$$

where a lower condition number indicates a less sensitive function.

For every test polynomial constructed, its condition number will be calculated and recorded. This will allow the standard assessment framework to determine how a root-finding algorithm behaves for both well-conditioned and ill-conditioned functions.

Number of Iterations and CPU Time

Measuring number of iterations is as straightforward as counting, but measuring CPU Time is slightly more complex.

Measuring CPU Time

Clearly, algorithms that are tested using the standard framework should be run on the same hardware, as it is easy to make one algorithm seem faster than another when it is run on a faster machine. However, even different implementations of the same algorithm, using either different languages or different coding techniques, can have vastly different performances, as shown by Kriegel et al. in their paper *The (black) art of runtime evaluation: Are we comparing algorithms or implementations?*, self-described as "…not a study to compare the efficiency of algorithms. Rather, this is a study on *comparing efficiency*" [4]. While the paper focuses on the methods used to measure the efficiency of data mining algorithms, mainly through the metric of runtime and space complexity, it provides excellent recommendations for researchers aiming to test any algorithm against others. Using those recommendations as a baseline, the standard assessment framework will be able to best measure CPU Time when the following guidelines are used:

- Ensure all algorithms to be tested are implemented in the same programming language

This eliminates the issue of some programming languages being inherently faster than others at certain tasks, masking an algorithm's true performance. Kriegel et al. recommend using a compiled language rather than an interpreted one if an algorithm contains a performance critical loop [4].

- Use the fastest possible implementations of the algorithm

Kriegel et al. also recommend creating multiple implementations and evaluating all of them to determine the fastest. They also highlight that the original implementation of another researcher's algorithm may not be the most optimal, and as such should also be compared against multiple implementations. Additionally, by publishing one's own algorithm source code online, they can encourage others to find other optimizations that they may have missed [4].

- Use standard libraries in implementations

The goal is to measure the CPU Time of a root-finding algorithm, not other tasks that may also occur when the algorithm is run. For example, if two algorithms that are being compared both involve inverting matrices, both implementations of the algorithms should use the same high-performance matrix inverting function from a standard math library, rather than custom ones.

- Use a standard runtime evaluation library

Almost all languages have a library that can be used to measure runtime; the same library should be used for all algorithms tested.

Method for Terminating an Algorithm

For factored form polynomials, the actual roots are known, so an algorithm can be terminated once the error between an approximation and the actual root is within a certain tolerance. For standard form polynomials, the actual roots are not known, so an algorithm can be terminated once the error between successive approximations is within a certain tolerance.

The tolerance values $\{1 \cdot 10^{-1}, 1 \cdot 10^{-2}, \cdots, 1 \cdot 10^{-16}\}$ will be used for each trial/sub-trial run on a specific constructed polynomial, where the last tolerance value is machine precision. Absolute error will be used, as this avoids a potential division by zero that could occur when relative error is used.

Convergence Order

The previously mentioned methods of calculating convergence order used in some papers all require symbolic manipulation of the algorithm itself, which is difficult to generalize to an unknown algorithm. To calculate the convergence order α for any algorithm, an empirical estimate, $\tilde{\alpha}$, can be used instead by examining the absolute errors, either between successive approximations when the actual root is not known, or between an approximation and the root when it is known. For a sequence $\{p_n\}_{n=0}^{\infty}$ of root approximations that converges to the actual root r, with $p_n \neq r$ for all n, when the actual root is known, we can use [10]

$$\tilde{\alpha} = \frac{\ln\left(\left|\frac{p_{k+1} - r}{p_k - r}\right|\right)}{\ln\left(\left|\frac{p_k - r}{p_{k-1} - r}\right|\right)}$$

whereas if the actual root is not known, we can use [10]

$$\tilde{\alpha} = \frac{\ln\left(\left|\frac{p_{k+1} - p_k}{p_k - p_{k-1}}\right|\right)}{\ln\left(\left|\frac{p_k - p_{k-1}}{p_{k-1} - p_{k-2}}\right|\right)}.$$

The value $\tilde{\alpha}$ can be calculated for all the ν trials/sub-trials run, with the results averaged to estimate a value for the true convergence order α:

$$\alpha \approx \frac{1}{\nu}\sum_{j=1}^{\nu} \tilde{\alpha}_j$$

The Theoretical Standard Testing Framework Process

Finally, with our test polynomials, metrics, and methods sorted out, we can describe how the proposed standard testing framework would test a root-finding algorithm.

1. Construct all of the test polynomials of various degrees, coefficient and root types, and multiplicities as described previously.
2. For each test polynomial:
    a. Calculate the following information:
        i. For all standard form polynomials, intervals containing all of the polynomial's roots.
            1. For standard form polynomials with entirely real roots, Laguerre proved that all roots will lie in the interval [7]

$$\left[-\frac{a_{n-1}}{n \cdot a_n} - \frac{n-1}{n \cdot a_n}\sqrt{a_{n-1}^2 - \frac{2n}{n-1} \cdot a_n \cdot a_{n-2}}, -\frac{a_{n-1}}{n \cdot a_n} + \frac{n-1}{n \cdot a_n}\sqrt{a_{n-1}^2 - \frac{2n}{n-1} \cdot a_n \cdot a_{n-2}}\right].$$

            2. For standard form polynomials with complex roots, Cauchy proved that all roots will lie within the disk [6]

$$|z| < 1 + max\left\{\left|\frac{a_{n-1}}{a_n}\right|, \left|\frac{a_{n-2}}{a_n}\right|, \cdots, \left|\frac{a_0}{a_n}\right|\right\}.$$

    b. Test the algorithm against the polynomial multiple times.
        i. For each test trial, run sub-trials, varying:
            1. The tolerance values used to determine when to terminate the algorithm.
            2. For each tolerance value, vary:
                a. The initial guess used to start the algorithm, randomly selecting initial guesses from within the bounds calculated previously.
        ii. For each sub-trial, record:
            1. The exact polynomial test function used.

    2. Its condition number.
    3. The tolerance value used.
    4. The initial guess used.
    5. The number of iterations and CPU time required for the algorithm to converge with the tolerance value, using the methods described previously.
    6. The convergence order estimate, using the method described previously.

3. Store all results.

A Proposed Implementation

To remove the "theoretical" from theoretical standard testing framework, one possible implementation of the framework could be as the form of freely available, open-source software, available to researchers and readers alike to test polynomial root-finding algorithms. Since a variety of metrics and information about the test polynomial used for each sub-trial run is stored, the results can be filtered according to the desires of the software's user.

For example, a reader may only care about whether an algorithm is accurate and fast, but not necessarily what types of polynomials it can run on; in this case, they can choose to look at results for all types of test polynomials, but filter those results to only look at sub-trials that used high tolerance values, and only display the CPU time measured instead of all of the metrics. On the other hand, another reader may wish to look at all possible metrics and sub-trial variations to gain the most accurate assessment of an algorithm, but only look at the sub-trials run on polynomials with real coefficients due to that reader's specific needs.

References

[1] Burden, R. L., Faires, J. D., & Burden, A. M. (2015). Numerical analysis. Cengage Learning.

[2] Burton, A. (2009). Newton’s Method and Fractals. Walla Walla; Whitman College.

[3] Hansen, E., & Patrick, M. L. (1976). A family of root finding methods. Numerische Mathematik, 27(3), 257–269. https://doi.org/10.1007/bf01396176

[4] Kriegel, H., Schubert, E., & Zimek, A. (2016). The (black) art of runtime evaluation: Are we comparing algorithms or implementations? Knowledge and Information Systems, 52(2), 341–378. https://doi.org/10.1007/s10115-016-1004-2

[5] Kurtz, D. C. (1992). A Sufficient Condition for All the Roots of a Polynomial To Be Real. American Mathematical Monthly, 99(3), 259. https://doi.org/10.2307/2325063

[6] Jain, V. (1990). On Cauchy’s bound for zeros of a polynomial. Approximation Theory and Its Applications, 6(4), 18–24. https://doi.org/10.1007/bf02836305

[7] Laguerre. Sur une méthode pour obtenir par approximation les racines d'une équation algébrique qui a toutes ses racines réelles. Nouvelles annales de mathématiques : journal des candidats aux écoles polytechnique et normale, Serie 2, Volume 19 (1880), pp. 161-171. http://www.numdam.org/item/NAM_1880_2_19__161_1/

[8] Lang, M., & Frenzel, B.-C. (1994). Polynomial Root Finding. IEEE Signal Processing Letters, 1(10), 141–143. https://doi.org/10.1109/97.329845

[9] Naseem, Amir, et al. “A Novel Root-Finding Algorithm With Engineering Applications and Its Dynamics via Computer Technology.” IEEE Access, vol. 10, 10 Feb. 2022, pp. 19677–19684, doi:10.1109/access.2022.3150775.

[10] Senning, J. R. (2020, August 7). Computing and Estimating the Rate of Convergence. Wenham; Department of Mathematics and Computer Science, Gordon College.

[11] Thota, S. (2019). A new Root–Finding algorithm using exponential Series. Ural Mathematical Journal, 5(1), 83–90. https://doi.org/10.15826/umj.2019.1.008

[12] Yuksel, C. (2022). High-Performance polynomial root finding for graphics. Proceedings of the ACM on Computer Graphics and Interactive Techniques, 5(3), 1–15. https://doi.org/10.1145/3543865